Database objects and programming methods

DB2 Information Management Software

`http://www-136.ibm.com/developerworks/db2`

Table of contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Before you start

What is this tutorial about?

This tutorial introduces different types of database objects and programming methods. In this tutorial, you will learn about:

- Different types of database objects
- The relationship between schemas and database objects
- The fundamentals of SQL routines
- The basics of dynamic and static embedded SQL
- The basics of CLI and ODBC
- The basics of ADO and OLE DB
- The basics of JDBC and SQLJ
- The security considerations associated with data access from an application

This is the first in a series of seven tutorials that you can use to help prepare for the DB2 UDB V8.1 Family Application Development Certification exam (Exam 703). The material in this tutorial primarily covers the objectives in Section 1 of the exam, entitled "Database objects and programming methods." You can view these objectives at: *http://www.ibm.com/certify/tests/obj703.shtml*.

You do not need a copy of DB2 Universal Database to complete this tutorial. However, you can download a free trial version of *IBM DB2 Universal Database* Enterprise Edition for reference.

Who should take this tutorial?

To take the DB2 UDB V8.1 Family Application Development exam, you must have already passed the DB2 UDB V8.1 Family Fundamentals exam (Exam 700). You can use the DB2 Family Fundamentals tutorial series (see Resources on page 25) to prepare for that exam. It is a very popular series that has helped many people understand the fundamentals of the DB2 family of products.

Although not all materials discussed in the Family Fundamentals tutorial series are required to understand the concepts described in this tutorial, you should have a basic knowledge of:

- DB2 instances
- Databases
- Database objects
- DB2 security

This tutorial is one of the tools that can help you prepare for Exam 703. You should also review the Resources on page 25 at the end of this tutorial for more information about database objects and programming methods.

About the author

Clara Liu works as a DB2 UDB consultant at the IBM Toronto Laboratory. As a member of the Data Management Channel Development Team, she works closely with IBM business partners. Clara specializes in database application development and integration of new technologies with DB2. She teaches DB2 UDB certification courses to IBM business partners and at conferences. She co-authored the book *DB2 SQL Procedural Language for Linux, UNIX, and Windows*. You can reach Clara at *claraliu@ca.ibm.com*.

Notices and trademarks

Section 2. DB2 database objects

Basic database objects

Database objects are the building blocks of a database. DB2 provides different types of database objects to store and represent different information. They can be created, modified, and dropped using *database definition language* (DDL). To manipulate them, you use *database manipulation language* (DML), such as `SELECT`, `UPDATE`, `INSERT`, and `SELECT` statements. Some commonly used database objects are:

- Tables
- User-defined data types
- Constraints
- Views
- Indexes

If you are not already familiar with these basic database objects, we recommend that you refer to the Family Fundamentals tutorial *Working with DB2 UDB objects* for background information.

In addition to the database objects introduced in the Family Fundamentals tutorials, there are other objects that many developers find useful when developing DB2 applications. We'll introduce these objects in this section.

One note before we proceed: In the examples that you'll see here, the object names are specified in lowercase. Regardless of the platform DB2 is running on, it will always store names in uppercase unless the name of an identifier is surrounded by double quotes ("").

For example, the following statement creates a table called *employee* (in lowercase) with the same column definitions as the table EMPLOYEE (in uppercase).

```
CREATE TABLE "employee" LIKE employee
```

Aliases

An *alias* is an alternative name for an existing table, view, or nickname. An alias can also serve as an alternative name for another alias. Like these objects, an alias can be created or dropped, and it can have comments associated with it. Here are a few examples of `CREATE ALIAS` statements:

```
CREATE ALIAS aliastab1 FOR tab1;
CREATE ALIAS bob.aliastab1 FOR tom.tab1;
CREATE SYNONYM bob.aliastab2 FOR bob.aliastab1;
```

As you can see, the `CREATE ALIAS` statement is straightforward. You can create an alias in the same schema as the source object (as in the first line) or fully qualify the alias name (as in the second line). It is also valid to use the keyword `SYNONYM` in place of `ALIAS` (as in the third line) for compatibility with DB2 for zSeries.

No special authority or privilege is required to use aliases. However, you need to obtain appropriate authorization associated with the underlying objects being referenced. For a complete listing of database object privileges, refer to the DB2 DBA certification tutorial *Server management* (see Resources on page25).

We mentioned earlier that aliases can also be created for *nicknames*. Nicknames are database objects that refer to data tables or views located on federated systems. Federated database support is outside of the scope of this tutorial. We'll learn more about federated systems in the second part of this tutorial series, *Data manipulation* (see Resources on page25 .

To add a comment to an alias, issue this statement:

```
COMMENT ON aliastab1 IS 'My first alias on tab1'
```

To drop an alias, use the `DROP` statement, as you would for any other database objects:

```
DROP ALIAS aliastab1
```

Sequence objects

A *sequence* is a database object that allows automatic generation of values. Unlike an identity column, which is bound to a specific table, a sequence is a global and stand-alone object that can be used by any table in the same database.

An identity column is a specific kind of sequence object. Therefore, the characteristics of identity columns also apply to sequence objects. An example of a `CREATE SEQUENCE` statement is illustrated here:

```
CREATE SEQUENCE myseq AS INTEGER
    START WITH 360
    INCREMENT BY 10
    NO MAXVALUE
    CYCLE
    CACHE 20
```

Any exact numeric data type with a scale of zero can be used for the sequence value. These include `SMALLINT`, `INTEGER`, `BIGINT`, or `DECIMAL`. Any user-defined distinct type based on these data types can also be used for sequence values. This further extends the use of user-defined distinct types in an application.

As shown in the example above, you can customize a sequence object by specifying its start value. In this example, the first value for the sequence is 360. The generation of subsequent values is controlled by the `INCREMENT BY` clause. Positive and negative constants are supported to produce ascending and descending sequence values.

By default, the smallest and largest values generated by a sequence are bounded by the limit of the sequence data type. For example, an `INTEGER` sequence value must be within the range from -2,147,483,647 to 2,147,483,647. You can find the limits of all the numeric data types in the DB2 SQL Reference Guide (see Resources on page 25). To change this behavior, use the `MINVALUE` and `MAXVALUE` options to set a boundary for the generated values. If the minimum or maximum value is reached, another option, called `CYCLE` or `NO CYCLE` can be used to specify whether the sequence values should be recycled. Note that when `CYCLE` is in effect, duplicate values can be generated for the sequence.

The `CACHE` option allows DB2 to keep some preallocated values in memory to improve performance. `CACHE 20` is the default behavior. There's one thing to keep in mind about this option: If DB2 is stopped before all cached values are used, any cached and unused values will be discarded. When DB2 is restarted, the next block of values is generated and cached, introducing gaps between values. If your application does not allow value gaps, consider using the `NOCACHE` option instead.

If caching is not used, performance could be substantially worse for high rate of sequence number generation. A log record gets written each time a new value is generated. As a result, it is more efficient to obtain the values in one request and cache them in memory.

You can change characteristics of a sequence object with the `ALTER SEQUENCE` statement. All the settings discussed above can be altered except for the data type of the sequence value. For the complete syntax diagram, refer to the DB2 SQL Reference Guide (see Resources on page25).

Dropping a sequence object is like dropping any other database object except that a `RESTRICT` keyword is required. This prevents the sequence from being dropped if any dependency exists.

```
DROP SEQUENCE myseq RESTRICT
```

Generating and retrieving sequence values

A sequence is a type of database object, therefore its access is also controlled by privileges. By default, only the sequence creator, SYSADM, and DBADM hold the USAGE privilege of the object. If you want other users to be able to use the sequence, you need to use the following statement:

```
GRANT USAGE ON SEQUENCE seq_object_name TO PUBLIC
```

Refer to the DB2 DBA certification tutorial *Server management* (see Resources on page 25) to learn more about DB2 security.

Two expressions are provided to generate and retrieve sequence values. NEXTVAL FOR *seq-name* is used to obtain the next sequence value, whereas PREVVAL FOR *seq-name* is used to retrieve the last sequence value generated. The examples below illustrate the use of these expressions.

```
INSERT INTO t1 VALUES (NEXTVAL FOR myseq, 'BOB');
INSERT INTO t1 VALUES (NEXTVAL FOR myseq, 'PAT');
COMMIT;
INSERT INTO t1 VALUES (NEXTVAL FOR myseq, 'GENE');
ROLLBACK;
INSERT INTO t1 VALUES (NEXTVAL FOR myseq, 'PAUL');
VALUES PREVVAL FOR myseq INTO :hostvar
```

Assume that we started with an empty table t1 and the next sequence value of myseq is 1. After the statements above are executed with autocommit disabled, t1 contains the following rows:

```
1 NAME
------- -------
1 BOB
2 PAT
4 PAUL

 3 record(s) selected.
```

Although the value generated for GENE was rolled back, DB2 will not reuse it. Thus, the next sequence value generated for PAUL is 4 rather than 3.

The last statement in the example shows how the PREVVAL expression can be used. The host variable :hostvar stores the last value generated in the current session. If you want to keep any value that was generated previously, you should save the PREVVAL value before generating the next one.

Temporary tables

As their name implies, *temporary tables* are not permanent database objects. A temporary table behaves like a normal table except that not all features and options are supported and/or required. It persists only for the duration of a connection. When the connection is closed, all temporary tables declared within it will be dropped automatically.

Only the session or application that declares a temporary table will have access to it. If two applications create a temporary table with the same name, each instance of the temporary table is unique. Thus, you do not need to worry about any temporary data collision scenario. Since a temporary table only allows single-connection access, locking is not required. This is one of the main performance benefits of temporary tables.

Declaring temporary tables

To declare temporary tables, a `USER` temporary tablespace (which is different from a `SYSTEM` temporary tablespace) must exist to store the definition and content of the temporary table. A `SYSTEM` temporary tablespace is only used by DB2 internally to perform operations such as sorts. The following simple statement creates a user temporary tablespace.

```
CREATE USER TEMPORARY TABLESPACE usertempspace
 MANAGED BY SYSTEM USING ('usertempspace')
```

A number of optional clauses are supported when declaring a global temporary table. The following examples illustrate their behavior.

```
DECLARE GLOBAL TEMPORARY TABLE t_dept
( deptid CHAR(6), deptname CHAR(20) )
ON COMMIT DELETE ROWS
NOT LOGGED
IN usertempspace
```

In this example, the temporary table t_dept is declared with two columns. The `ON COMMIT DELETE ROWS` clause forces the content of the temporary table to be deleted at every `COMMIT` operation. In DB2 V8, you may choose to log changes on the temporary table for the purpose of rollback. This example specifies that changes to the table are `NOT LOGGED`. This means that any manipulation of the table, including creation and changes of the table, are not logged. If the table is created in a unit of work and is then rolled back, the temporary table will be dropped. On the other hand, if the table is dropped in the unit of work, the table will be restored without any rows.

You do not need to use the `IN` clause to specify the user temporary tablespace that this temporary table is going to use. If you don't specify this information, DB2 will search for the most suitable tablespace. If a user temporary tablespace cannot be found, DB2 will raise an error.

Let's look at another example:

```
DECLARE GLOBAL TEMPORARY TABLE t_proj
    LIKE project
```

```
      ON COMMIT PRESERVE ROWS
      WITH REPLACE
      IN usertempspace
```

The temporary table t_proj is declared with the `LIKE` keyword so it has the same column definition as the persistent table or view named *project*. The `ON COMMIT PRESERVE ROWS` clause indicates that all rows in the temporary table will be preserved at `COMMIT`. Thus, they will be available for further processing in the following transaction.

Before you can declare another temporary table with the same name in the same session, the temporary table has to be dropped. This can be done explicitly or with the `WITH REPLACE` option as shown here. With the `WITH REPLACE` option, DB2 implicitly deletes all data, drops the temporary table, and re-creates it with the new definitions.

The `WITH REPLACE` option becomes very handy when *connection pooling* is used. Connection pooling is a mechanism for reusing database connections so that resources are not allocated and deallocated on demand. These operations are quite costly, especially when there are large numbers of connections that perform relatively short transactions. Since connections are not released, it is possible that previously used temporary tables are not cleaned up. The next application to use the connection may pick up data left from the previous execution. Therefore, using the `WITH REPLACE` option guarantees that the temporary table to be declared is refreshed with the new definition.

Schemas and database objects

Most database objects are identified by a *schema* and an *object name*. A database schema provides a logical classification of database objects. Examples of the two-part object names are illustrated here:

```
  DB2ADMIN.EMPLOYEE
  HRPROD.AUDIT_VIEW
  HRPROD.AUDIT_TRIG
```

If a schema is not specified when accessing the database object, the user ID used to establish the database connection will be set as the default schema. For example, if the user db2admin connects to a database and creates a table T1, DB2 will create a table called db2admin.T1. All the subsequent SQL statements that referenced an unqualified table T1 are resolved to db2admin.T1.

DB2 special register: CURRENT SCHEMA

As you may have already discovered in your database work, it is not always feasible to connect to a database with the ID of the user who is also the schema of the objects. Hard-coding an application to fully qualify the objects is also not the best solution. Fortunately, DB2 allows you to change the current schema using the `SET CURRENT SCHEMA` command. By default, the `CURRENT SCHEMA` DB2 special register is set to the `USER` connected to the database. When you change the `CURRENT SCHEMA`, any unqualified database objects are prefixed with the new value.

The current schema can be obtained with this command:

```
VALUES CURRENT SCHEMA
```

To change it, simply use this command:

```
SET SCHEMA=db2admin
```

Section 3. DB2 application development database objects

Routines

In this section, we introduce a few more types of database objects. These objects are collectively known as *routines*. There are three main functional types of routines: *stored procedures*, *functions*, and *methods*. (For more on the differences between these types, see Stored procedures, functions, and methods on page12 .)

Routines are database objects that encapsulate programming and database logic related to a specific task. Using routines effectively can simplify your application code and increase code reusability. For example, if certain business logic is encapsulated in a routine, changes to the business rules will only affect that particular routine. As a result, application changes can be minimized.

Routines are defined and processed at the database server. This allows an application to exploit the power of the database server while reducing the processing load required at the client.

Typically, complex business logic issues multiple SQL statements. Statements are sent separately from the client machine to the server, resulting in high network traffic if heavy database activities are involved. If these operations are captured in a routine, network traffic between the client and server machines can be reduced significantly, thus improving overall application performance.

The ability to tighten security is another key advantage of using routines. Routines can be used to mask direct access to the underlying database objects. It is sufficient to invoke a routine with `EXECUTE` privilege; no explicit privileges for the underlying database objects are required.

Different implementations of routines

A few implementations for routines are possible:

- *Built-in* routines come with the DB2 system. They are defined in system schemas such as `SYSIBM`, `SYSPROC`, `SYSFUN`, and `SYSTOOLS`.
- *Sourced* routines only apply to functions. A sourced function duplicates the semantics of another function.
- *External* routines are implemented in an external programming language. As of DB2 V8.1, you can develop routines in the following supported languages:
  - The Java language
  - C/C++
  - OLE (for stored procedures only)

    - OLE DB (for table functions only)
- *SQL* routines are implemented in the SQL Procedural Language (SQL PL). DB2 SQL PL is a subset of the SQL Persistent Stored Modules (SQL/PSM) language standard. This standard is the basis for the structured programming languages used with SQL to write stored procedures, functions, and methods. It combines the ease of data access of SQL with the flow control structures of a simple programming language. This is the main reason why SQL PL is so popular.

Stored procedures, functions, and methods

Stored procedures are database objects that contain specialized programs to access and modify data in one or more tables. Within a stored procedure, you can wrap multiple SQL statements with flow logic. They serve as subroutine extensions to client applications or other routines. They are executed and managed under the control of a relational database management system (RDBMS).

Functions are SQL extensions that you can customize. They are invoked from within SQL statements such as a `select-list` or a `FROM` clause. There are four types of functions: *aggregate*, *scalar*, *row*, and *table*.

Stored procedures are typically used to encapsulate complex application logic and perform heavy database operations such as multi-table joins and cursor manipulations. On the other hand, functions -- especially SQL functions -- usually contain simpler operations. Statements coded in SQL functions are expanded into, and executed with, the SQL statements that reference them. This causes the statements to be dynamically compiled at execution time. If a complex SQL function is used in an SQL statement, DB2 requires additional resources to compile it and generate a data access plan. This may affect overall run-time performance.

A *method* encapsulates logic that provides behavior for structured types that contain one or more named attributes, each with its own data type.

DB2 special register: CURRENT PATH

Like most DB2 database objects, a complete routine name is composed of a schema and a routine name. The DB2 special register introduced earlier, `CURRENT SCHEMA` (see DB2 special register: CURRENT SCHEMA on page 9), resolves schemas for most database objects but not for routines. Instead, DB2 locates a routine with the use of the `CURRENT PATH` special register.

You can obtain the `CURRENT PATH` setting with this command:

```
VALUES CURRENT PATH
```

The default path contains three system schemas followed by the current user connected to the database. For example:

```
"SYSIBM","SYSFUN","SYSPROC","DB2ADMIN"
```

Here are a few examples and tricks you can use to update the `PATH`:

```
SET PATH=CURRENT PATH,"USER2"
VALUES CURRENT PATH
------------------------------
"SYSIBM","SYSFUN","SYSPROC","DB2ADMIN","USER2"
1 record(s) selected.


SET PATH=SYSTEM PATH,"USER2"
VALUES CURRENT PATH
------------------------------
"SYSIBM","SYSFUN","SYSPROC","USER2"
1 record(s) selected.


SET PATH="USER1","USER2","USER3"
VALUES CURRENT PATH
------------------------------
"USER1","USER2","USER3"
1 record(s) selected.
```

Section 4. Embedded SQL programs

An introduction to embedded SQL

DB2 provides different programming methods for developers to write applications. One of the most direct approaches is embedded SQL programming. This programming method is *direct* because the program is embedded with SQL statements that interact directly with DB2. Programs can be written in any of the following supported programming languages:

- C/C++
- FORTRAN
- COBOL
- The Java language (SQLJ)

How you structure your embedded SQL statements depends on the programming language you choose. Embedded SQL statements in C/C++ and FORTRAN are prefixed with the `EXEC SQL` keywords:

```
EXEC SQL SELECT lastname,empid INTO :hostvar1, :hostvar2
        FROM employee WHERE deptno="OPERATIONS";
```

Embedded SQL statements in COBOL are very similar to those coded in C/C++, except the keyword `END-EXEC` is required at the end of the statement:

```
EXEC SQL SELECT lastname,empid INTO :hostvar1, :hostvar2
           FROM employee WHERE deptno="OPERATIONS" END-EXEC;
```

Embedded SQL statements in the Java language are slightly different than those shown in the previous two examples in that the connection context in which the statement is going to be executed must be specified. Here is an example:

```
#sql [myConnCtx] {SELECT lastname, empid INTO :hostvar1, :hostvar2
                    FROM employee WHERE deptno="OPERATIONS"};
```

To give you a better idea of what an embedded SQL program looks like, here is a snippet of such a program written in C. In the third tutorial in this series (see Resources on page 25), you will learn more about program precompilation and actual code development.

```
int TbBasic(void)
{
  int rc = 0;
  struct sqlca sqlca;

  EXEC SQL BEGIN DECLARE SECTION;
```

```
      char hostVarStmt[50];
      EXEC SQL END DECLARE SECTION;

    /* delcare cursor */
    EXEC SQL DECLARE c1 CURSOR FOR
      SELECT deptnumb, deptname FROM org WHERE deptnumb = 40;

    /* open cursor */
    EXEC SQL OPEN c1;

    /* fetch cursor */
    EXEC SQL FETCH c1 INTO :deptnumb, :deptname;

    while (sqlca.sqlcode != 100)
    {
      printf("    %8d %-14s\n", deptnumb, deptname);
      EXEC SQL FETCH c1 INTO :deptnumb, :deptname;
    }

    /* close cursor */
    EXEC SQL CLOSE c1;

    /* prepare the statement */
    strcpy(hostVarStmt, "DELETE FROM org WHERE deptnumb = 15");
    EXEC SQL PREPARE Stmt FROM :hostVarStmt;

    /* execute the statement */
    EXEC SQL EXECUTE Stmt;

    /* ROLLBACK the transaction */
    EXEC SQL ROLLBACK;

    return 0;
  }
```

Static SQL

In the previous panel, we demonstrated two different types of embedded SQL statements: static embedded SQL and dynamic embedded SQL.

The `SELECT` statement used in the example illustrated the use of static SQL. To write static SQL, the complete statement must be specified. The names of the tables and columns as well as the data types being referenced are known. The only information that can be specified at run time are values in the `WHERE` clause of the SQL statement using host variables.

When a static SQL statement is compiled (or, more accurately in DB2 terminology, *prepared*), its data access plan is generated and stored in a package in the database. This package is executed when the application containing the statement is invoked. Since no statement compilation is required at run time, there is no compilation cost to execute static SQL.

DB2 uses database statistics and configuration parameters to evaluate and obtain an access plan when the statement is being prepared. As the database statistics change, the pregenerated access plan may not be as optimal as it was when initially generated.

The authorization of the person who prepares a static SQL statement is used to create and bind the package to the database. As long as the person who executes the package has `EXECUTE` privilege for the package, this person does not require explicit privileges on the database objects referenced in the package. See Security considerations on page 16 for a more detailed discussion of DB2 privileges.

Dynamic SQL

Dynamic SQL statements are processed at run time dynamically. The statement structure is not required until the application is executed.

Note that the example SQL statement shown in An introduction to embedded SQL on page 14 uses dynamic SQL to `PREPARE` and `EXECUTE` a `DELETE` statement. The text of the `DELETE` statement is stored in a host variable, `hostVarStmt`. When the statement is `PREPARE`d, an executable form of the statement is created and stored in a package in the database. Once the data access plan is generated, the prepared statement can be `EXECUTE`d. This does sound familiar, doesn't it? That's right: These two processing phases are exactly the same for static SQL statements. The only difference is that dynamic SQL processes both `PREPARE` and `EXECUTE` at run time, whereas static SQL prepares an access plan and keeps it in the database at precompile time.

Dynamic SQL statements must always be prepared before execution, regardless of whether the same statements (and hence the same access plans) are used over and over again. To minimize the cost of these prepares, DB2 provides a *package cache* (also known as *dynamic query cache*) to keep frequently used access plans in memory. The package cache significantly reduces the cost of repeated SQL prepare requests, but there is still the overhead of issuing and responding to prepare requests.

Current database statistics are always used during the preparation of the statement. As a result, the most optimal data access path can be generated.

Authorization of the SQL statement is determined at run time. The person who executes the application must have proper privileges to access the database objects referenced in the statements. The next panel provides a more detailed DB2 privileges discussion.

Security considerations

To precompile an embedded static SQL program, the user needs explicit privileges on the database objects referenced in the program. Since the precompile or `PREPARE` phase generates and stores a package in the database, this user must also have the `BINDADD` privilege to add new packages to the database. To execute a program with static SQL, the user simply needs `EXECUTE` privilege on the associated packages.

A user who compiles embedded dynamic SQL programs also needs `BINDADD` privilege. No other privilege is required because dynamic SQL is not prepared at compile time. At program execution time, the user requires all the necessary privileges to issue each SQL statement as well as `EXECUTE` privilege on the packages created for the embedded SQL programs.

The following table summarizes privileges required to precompile and execute a program with only static SQL or only dynamic SQL statements.

Role	**Privileges required for static SQL**	**Privileges required for dynamic SQL**
Developer (precompiles the program)	`BINDADD` privilege to add new packages to the database; explicit privileges on the database objects	`BINDADD` privilege to add new packages to the database
Application user (executes the program)	`EXECUTE` privilege to execute packages associated with the program	Explicit privileges on the database objects; `EXECUTE` privilege to execute packages associated with the program

Static SQL vs. dynamic SQL

Let's summarize and expand on what we've learned about static and dynamic SQL in the following table:

Static SQL	**Dynamic SQL**
Structure of the SQL statement is known. References to columns, tables, and data types must be specified. Variables can only be used for search values in the `WHERE` clause of the statements.	Statement structure is not required at precompile time. The statement text itself can be stored in a variable that will be referenced during application execution. This provides flexibility to support programs because the final form of the statement is not known at application design time.
Data access plan is generated at precompile time. The package is persistent and reusable in the database. If the statements are run only occasionally, static SQL may give you better run-time performance than dynamic SQL.	Data access plan is generated at run time. It is stored in a memory location called a *package cache*. If the same statement is executed over and over again, the in-memory data access plan generated in the `PREPARE` phase can be reused. Thus, dynamic SQL does not necessarily perform worse than static SQL. However, dynamic SQL executed for the first time must incur the full cost of the `PREPARE` phase.

Database statistics and configuration parameters at precompile time are used to generate an optimal data access plan.	Database statistics and configuration parameters at run time are used to generate an optimal data access plan. If `RUNSTATS` is performed frequently, dynamic SQL can take advantage of the latest and greatest database statistics.
Authorization is validated at precompile time. Developers who precompile programs with static SQL must have proper authorizations and privileges to access objects referenced in the statements. Once the packages are created, users who execute them require `EXECUTE` privilege for the packages.	Authorization is validated at run time. Users who execute programs with dynamic SQL must have proper authorizations and privileges on the objects referenced in the statements.

Section 5. Driver-based APIs

Driver-based APIs

Although most database vendors support embedded SQL database access, some additional work is required if you want to use the same source code and deploy embedded SQL applications to multiple database systems. Programs must be precompiled with different vendors' specific precompilers. Data access plans must also be generated and bound to the target databases. To increase the portability of your applications, you may want to consider using one of the *driver-based solutions* introduced in this section.

A driver-based solution involves a *driver manager* with which the application interfaces. The driver manager provides a set of industry standard application programming interfaces (APIs) to access a data source. An application is written to call these APIs and then compiled and linked with the manager's libraries. Because the drivers comply to the standard, an application can easily access different vendors' data source by using (or loading) the correct driver.

In the next few panels, we'll look at some of the APIs that DB2 supports. You will learn about them in detail in later tutorials in this series.

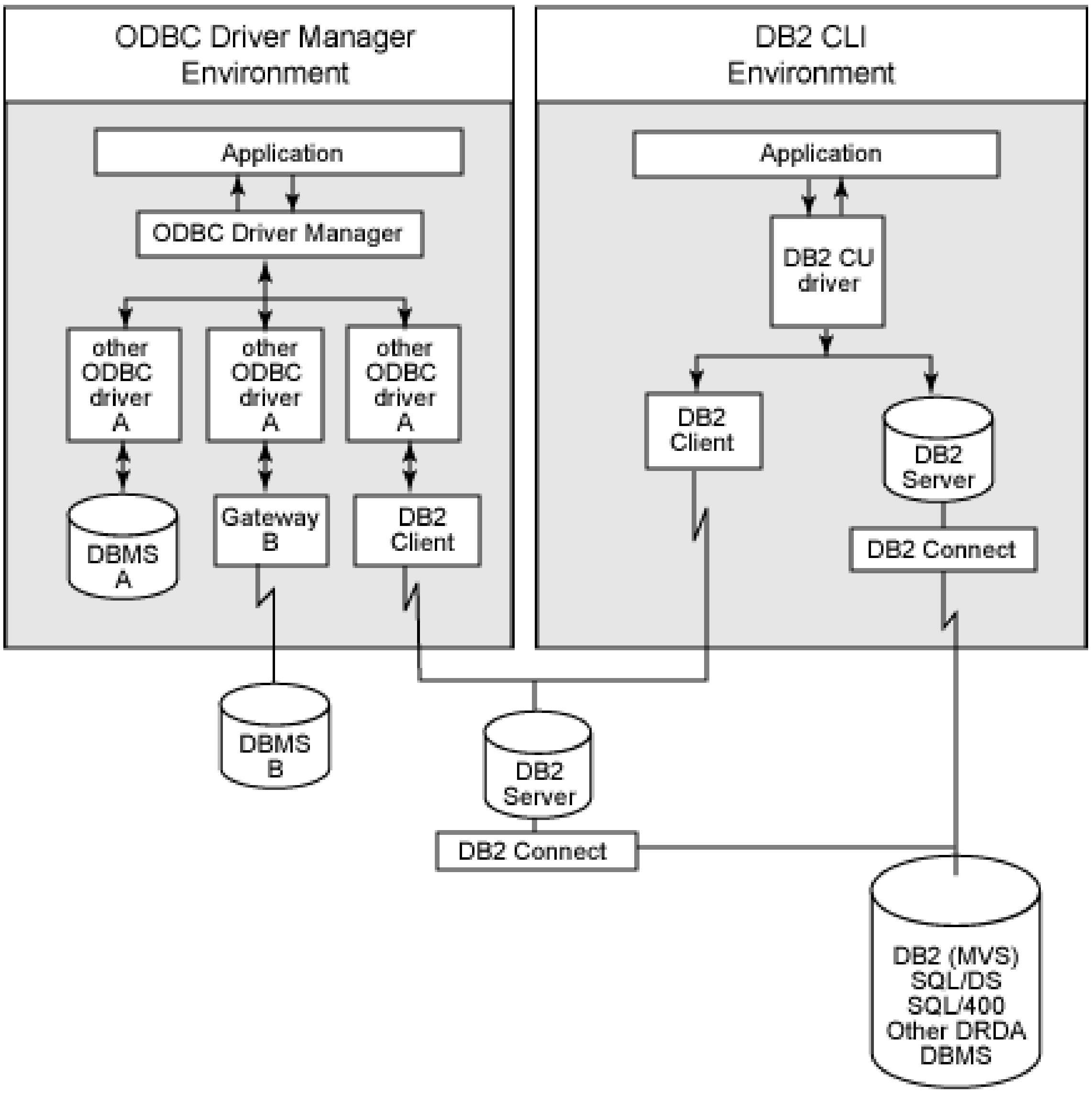
CLI and ODBC APIs

The DB2 Call Level Interface (CLI) is the IBM callable SQL interface to DB2 database servers. It is a C/C++ application programming interface (API) for database access. You can develop dynamic applications using DB2 CLI just as you can program with embedded dynamic SQL statements. In both cases, SQL statements are prepared and processed at run time. While embedded dynamic SQL requires a precompiler, DB2 CLI does not. You simply compile and link your application with the DB2 CLI driver libraries.

DB2 CLI is based on the Microsoft Open Database Connectivity (ODBC) and the X/Open CLI standards. In an ODBC environment, an application interfaces with the database server via an ODBC driver manager. The manager dynamically loads the necessary driver for the database server to which the application connects. ODBC provides several levels of function support, and the DB2 CLI currently conforms to ODBC 3.51.

To develop DB2 CLI applications, you need the DB2 Application Development Client. It includes all the necessary header files and libraries for compiling and linking CLI applications. On the other hand, to develop ODBC applications, you need an ODBC Developer's Software Kit supplied by Microsoft.

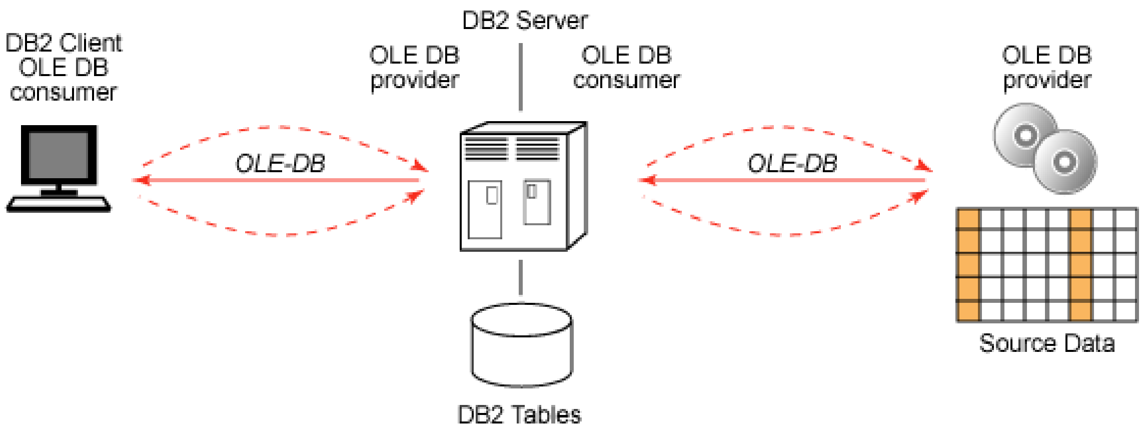
The figure below offers a quick comparison between DB2 CLI and ODBC environments.



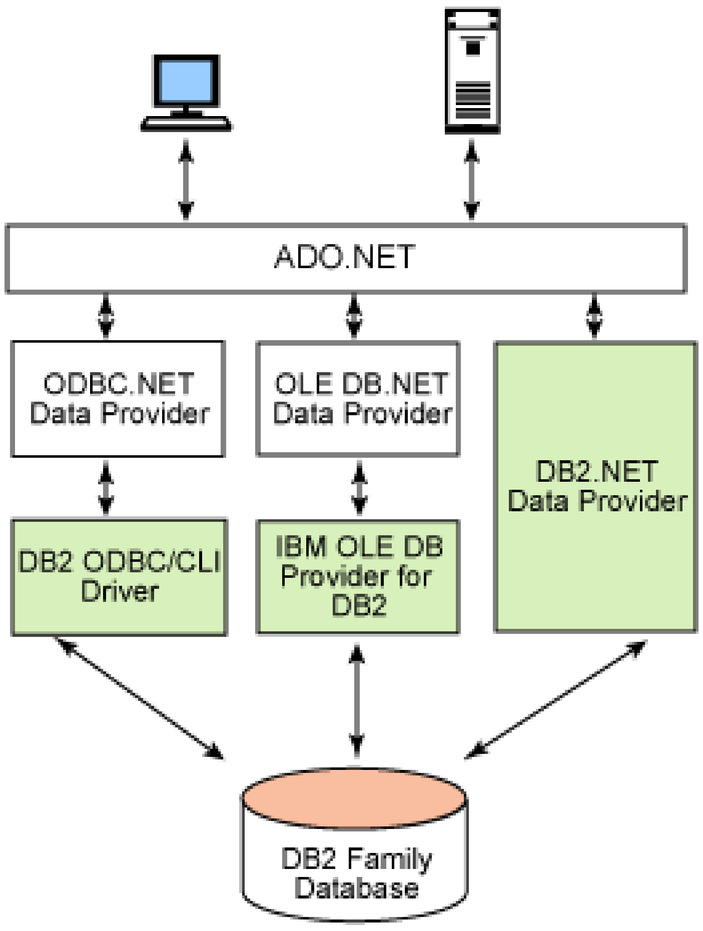
OLE DB and ADO.NET APIs

Microsoft Object Link Embedded Database (OLE DB) is a set of interfaces that provides applications with uniform access to data stored in diverse information sources, including relational and non-relational data sources. The OLE DB architecture consists of OLE DB providers and OLE DB consumers. DB2 supports both roles. The IBM OLE DB Provider for DB2 is required to enable DB2 to act as an OLE DB provider. This is a native driver that enables the ability to extract or query DB2 data through the OLE interface.

The figure below shows that a DB2 server can serve as an OLE DB consumer to access data from any OLE DB-compliant data source, such as a spreadsheet. The same DB2 server can also act as an OLE DB provider to supply data to an OLE DB consumer.



ActiveX Data Object .NET (ADO.NET) is another data access application interface Microsoft provides. It is a new data access model that combines the best features of its predecessors (ADO), together with traditional database access, integrated XML support, and a disconnected data architecture. There are three different access methods to connect to a DB2 database from a .NET application, as illustrated in the figure below.



JDBC and the SQLJ API

The Java platform has been gaining popularity for quite some time now. There are still many ongoing efforts to extend its functionality and performance. Java programs that access and manipulate DB2 can use the Java Database Connectivity (JDBC) API, along with the Embedded SQL for Java (SQLJ) standard. Both options are vendor-neutral SQL interfaces that provide data access to your application through standardized Java methods.

JDBC is the *de facto* standard Java API for relational database access using dynamic SQL with a powerful object-oriented interface. JDBC passes dynamic SQL to a JDBC driver that comes with DB2. DB2 executes the SQL statements through the JDBC APIs, and results are passed back to your Java code. JDBC is similar to DB2 CLI in that you do not have to precompile or bind a JDBC program, since JDBC uses dynamic SQL.

SQLJ programs, in other hand, contain static embedded SQL statements. Steps similar to precompile and bind are required to prepare an SQLJ program. Before you can compile an SQLJ source file, you must translate it with the SQLJ translator to create native Java source code. After translation, you need to create the DB2 packages using the DB2 for Java profile customizer (`db2profc`). A complete discussion of how to build SQLJ applications is covered in the fifth tutorial in this series, *Java programming*.

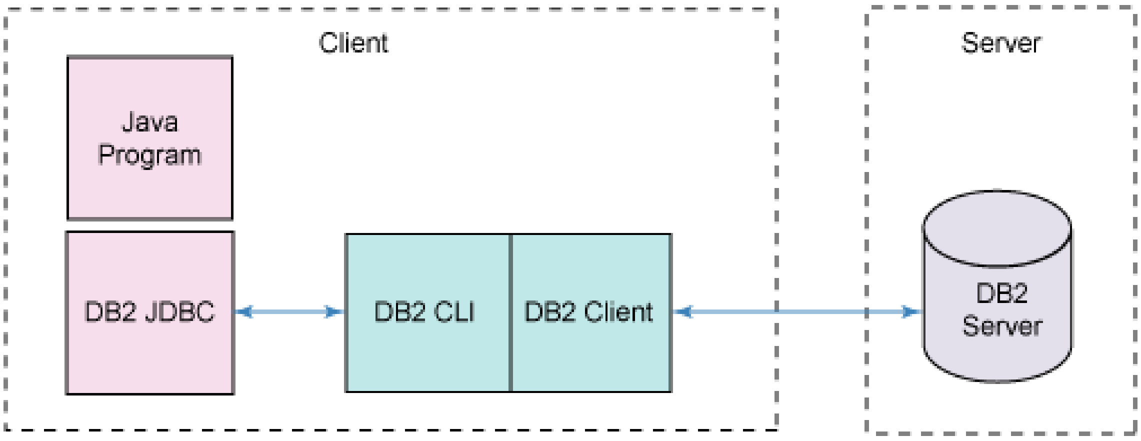
DB2 legacy CLI-based JDBC driver

According to the Java 2 Platform, Enterprise Edition (J2EE) specification, Java methods and interfaces used to gain data access can be packaged as JDBC drivers. DB2 JDBC drivers are categorized as legacy CLI-based drivers and new universal JDBC drivers.

The legacy CLI-based JDBC driver is built on top of the native DB2 CLI. There are two implementation classes. The first is a JDBC Type 2 driver, also known as an *app driver*. It is implemented by the class `com.ibm.db2.jdbc.app.DB2Driver`. The other implementation is a JDBC Type 3 driver, also known as a *net driver*.

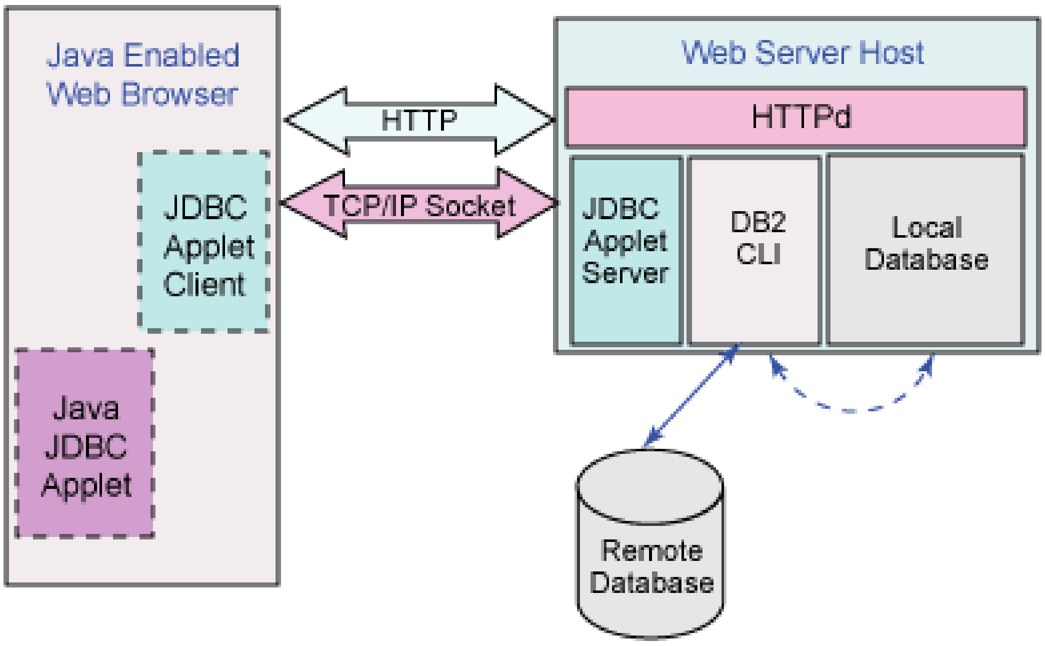
To use the app driver, a DB2 client must be installed on the client machine where the JDBC programs are to be executed. All the JDBC calls are translated into DB2 CLI calls.

DB2 JDBC Type 2 app driver



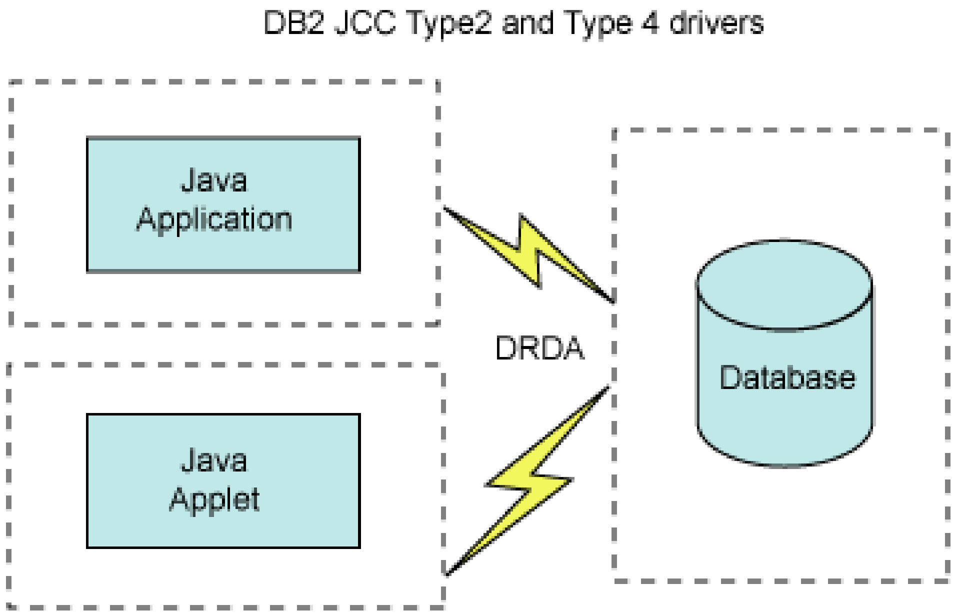
The DB2 JDBC Type 3 driver also maps JDBC calls to DB2 CLI. With this driver, a DB2 client is not required on the client machine. However, a JDBC listener (or JDBC daemon) is needed. The net driver is deprecated in DB2 V8; migration to a Type 4 driver is recommended. We'll discuss a Type 4 driver in the next panel.

DB2 JDBC Type 3 net driver



DB2 Universal JDBC driver

Fixpak 2 of DB2 V8 introduced the DB2 Universal JDBC driver. It is called a *universal* driver because it uses the Distributed Relational Database Architecture (DRDA) protocol that is common to all DB2 family databases. This driver enables direct Java connectivity to DB2 servers using a Type 4 driver architecture. The driver also has two implementation classes, called the Java Common Client (JCC) Type 2 driver and JCC Type 4 driver. The implementation class name is `com.ibm.db2.jcc.DB2Driver`.



Section 6. Conclusion

Summary

This tutorial introduced you to the basics of DB2 application development. Prior to any development work, you need to be familiar with different types of database objects. In this tutorial, several advanced database objects were introduced. Aliases, sequence objects, and temporary tables are used frequently to simplify application code. Stored procedures, functions, and methods are mainly used to encapsulate programming and database logic. Several advantages of using routines were discussed, such as the reduction of network traffic, the tightening of security, and an improvement of the reusability of code.

You should choose to encapsulate logic in procedures and functions based on the complexity of your application logic. Stored procedures serve as subroutines of programs and typically contain complex application logic. User-defined functions, on the other hand, are extensions to SQL. They perform very efficiently with simple statements and logic.

We also introduced a few programming methods in this tutorial. Embedded SQL programming allows SQL statements to be embedded in programs written in C/C++, FORTRAN, COBOL, or the Java language. Characteristics of static and dynamic SQL code were highlighted. You need to understand the differences in order to choose between the use of static and dynamic SQL statements.

Application programming interfaces such as CLI, ODBC, ADO, OLE DB, and JDBC are all dynamic programs. DB2 provides native drivers enabling applications to work with DB2 data through these interfaces. SQLJ, on the other hand, is a kind of static program written in the Java language. We also discussed the basics of how applications interface with DB2 using these APIs.

With the DB2 application development fundamentals introduced in this tutorial, you are now ready to learn more in-depth information about different development methods.

Resources

- For more information on the DB2 UDB V8.1 Family Application Development Certification exam (Exam 703), see *IBM DB2 Information Management -- Training and certification* (http://www.ibm.com/software/data/education/) for information on classes, certifications available and additional resources.
- As mentioned earlier, this tutorial is just one in a series of seven tutorials designed to help you prepare for the DB2 UDB V8.1 Family Application Development Certification exam (Exam 703). The complete list of all tutorials in this series is provided below:

1. Database objects and Programming Methods
2. *Data Manipulation*
3. *Embedded SQL Programming*
4. *ODBC/CLI Programming*
5. *Java Programming*
6. *Advanced Programming*
7. User-Defined Routines

- Before you take the DB2 UDB V8.1 Application Development certification exam (Exam 703), you should have already taken and passed the DB2 V8.1 Family Fundamentals certification exam (Exam 700). Use the *DB2 V8.1 Family Fundamentals certification prep tutorial series* to prepare for that exam. A set of six tutorials covers the following topics:
  - DB2 planning
  - DB2 security
  - Accessing DB2 UDB data
  - Working with DB2 UDB data
  - Working with DB2 UDB objects
  - Data concurrency
- Use the *DB2 V8.1 Database Administration certification prep tutorial series* to prepare for the DB2 UDB V8.1 for Linux, UNIX and Windows Database Administration certification exam (Exam 701). A set of six tutorials covers the following topics:
  - Server management
  - Data placement
  - Database access
  - Monitoring DB2 activity
  - DB2 utilities
  - Backup and recovery
- Check out *developerWorks Subscription* for one-stop access to a comprehensive portfolio of the latest IBM software from DB2, Lotus, Rational, Tivoli, and WebSphere, allowing you to maximize ROI and lower your labor costs, leading to superior productivity.
- You can learn more about database objects and client and server application development from the *DB2 Information Center*. In particular, look to these sections:
  - DB2 Version 8 Administration Guide: Implementation
  - DB2 Version 8 SQL Reference Guide
  - DB2 Version 8 Application Development Guide: Programming Client Applications
  - DB2 Version 8 Application Development Guide: Programming Server Applications
- For more information on DB2 Procedural Language, see *DB2 SQL Procedural Language for Linux, UNIX, and Windows*, Paul Yip et al. (International Business Machines Corporation, 2003; ISBN: 0-13-100772-6).
- Check out *developerWorks Subscription* for one-stop access to a comprehensive portfolio of the latest IBM software from DB2, Lotus,

Rational, Tivoli, and WebSphere, allowing you to maximize ROI and lower your labor costs, leading to superior productivity.

Feedback

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

For more information about the Toot-O-Matic, visit www-106.ibm.com/developerworks/xml/library/x-toot/ .